Programming Assignment Report - #2

Yonwoo Choi (2022-28614)
yhugestar@snu.ac.kr
Seoul National University - Advanced Graphics (Spring 2023)

Abstract

This programming assignment focuses on the development of a simulation framework for soft and rigid objects, encompassing fundamental setups and interactions. The main goal is to create a realistic environment with accurate collision handling and interactive capabilities.

In step 1, I have implemented of a basic simulator with essential functions such as starting, pausing, and resetting the simulation, as well as adding soft or rigid objects. For soft objects, the volume preservation correction technique is applied to ensure shape preservation. An example soft object, the bunny, is included, along with a rigid ball. A boundary is established to confine the objects within a cubic space. The TimeStepSize and NumSubSteps sliders control the time-step and the number of sub-steps in the Position-Based Dynamics (PBD) algorithm, and their impact on the simulation is reported.

For step 2 and 3, I focused on enhancing the simulation with collision handling and mouse interaction. Collision handling prevents object penetration by extending the techniques used for boundary handling. Mouse interaction enables the creation of attachment constraints between the closest object and the mouse pointer, allowing for interactive manipulation. I have created a scene using the framework, and a video showcasing the scene is recorded.

For the optional problem, I have implemented spatial hashing for collision detection. This reduces the time complexity from $O(n^2)$ to sub-linear time. A time comparison analysis is conducted to demonstrate the benefits of spatial hashing for different population sizes.

1 Basic Setups for Soft and Rigid Objects

1.1 Implementation of Basic Simulator

Instead of using the provided base Javascript, Pyscript code, implementation was fully done using Python's Open3D library. Open3D library does not provide built-in support for buttons and sliders in the visualization window, so the buttons were replaced by keyboard input to simulate button presses. The registered key callbacks are mapped as the table below.

Key	Key Callback
S	Start
P	Pause
R	Reset
A	Add soft object
B	Add rigid object

The simulation can be started, paused, and reset. When a rigid object is added to the scene, a sphere (ball) is added to the scene. Since a rigid object is a solid object with a fixed shape and volume, there is no need for volume correction, so only collisions are handled in the simulation. A custom **cube.json** file has been created for the soft object. The file has the verticies and triangle faces of the cube.

For a soft object, volume correction is necessary because the object needs to maintain its intended volume over time. Without volume correction, the object can expand or contract, which is far from realistic behavior. [1] Applying the volume correction, the soft object's vertices are adjusted based on the discrepancy between the current volume and initial volume. This correction helps to maintain the object's shape and volume integrity though out the simulation. *Algorithm 1* shows the steps of volume correction for soft objects.

As shown in Figure 6, a boundary cube of size **10m x 10m x 10m** is created in the scene. All the simulated objects do not go out of the boundary by clamping all the vertices that are about the leave the boundary.

```
function volumeCorrection(softObject):
    Input  : softObject: The soft object to perform volume
             correction on
    Output: The soft object with corrected volume
    initialVolume ← calculateVolume(softObject);
    currentVolume ← calculateVolume(softObject);
    correctionFactor ← $\left(\frac{\text{initialVolume}}{\text{currentVolume}}\right)^{\frac{1}{3}}$;
    centerOfMass ←
      calculateCenterOfMass(softObject.vertices);
    for vertex in softObject.vertices do
        vertex ← centerOfMass + (vertex - centerOfMass) *
          correctionFactor;
    end
    return softObject;
end
```

Algorithm 1: Volume Correction of Soft Objects

1.2 Time step, Sub steps

The timestep size determines the duration of each iteration in the simulation. Increasing the timestep size allows the simulation to progress more quickly, as each iteration covers a larger time interval. Conversely, decreasing the timestep size slows down the simulation, as each iteration represents a smaller time interval. A smaller timestep size can provide more accurate results by capturing finer temporal details, but it may come at the cost of increased computational resources.

On the other hand, the number of substeps determines the granularity of updates within each main iteration of the simulation loop.[2] By increasing the number of substeps, the object positions and velocities are updated more frequently within each main iteration. This finer-grained approach enables better representation of fast or complex interactions and can enhance the overall accuracy of the simulation. However, a higher number of substeps also requires more computational resources.

Adjusting the timestep size and the number of substeps provides a trade-off between simulation speed and accuracy. Finding the appropriate balance depends on the specific requirements of the simulation, considering factors such as the desired level of precision, computational resources available, and the nature of the simulated phenomena.

The sliders for adjusting the time step size and the number of sub steps was replaced by the keyboard input. The registered key callbacks are shown in the table below.

Key	Key Callback
1	Decrease time step size
2	Increase time step size
3	Decrease number of sub steps
4	Add Increase number of sub steps

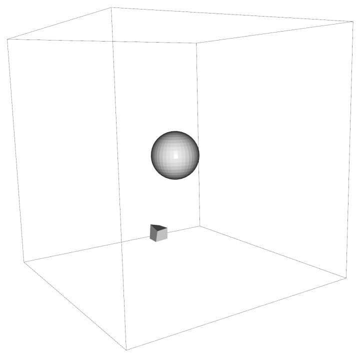

Figure 1: Basic Simulator with Boundary

2 More Interactions

2.1 Collision handling between objects

Collision handling in this simulation is divided into two main parts: Collision Detection and Collision Response. Collision detection between pairs of rigid objects is performed by calculating the distance between their centers. There is an assumption that the objects are af equal mass and perfectly spherical. If the distance is less than the sum of their radius, a collision is detected. When a collision between two rigid objects is detected, their velocities along the direction of the collision are swapped. This simulates a perfectly elastic collision between the two objects.

For detecting collision between soft objects, each vertex of the first soft body against each triangle of the second object is checked. For detecting collision, the simulation uses the Gilbert-Johnson-Keerthi algorithm [3] to determine whether two convex sets intersect. Once a collision is detected, the response involes a deformation of the soft object by applying a force to the vertices of the objects that are involved in the collision. After the deformation, the soft object could be potentially squished and its volume might change so the correct this, a volume conservation method is applied by scaling the verticies of the soft object such that its volume retains consistent.

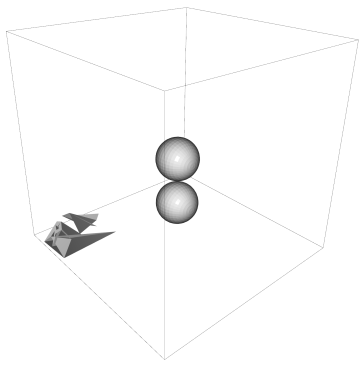

Figure 2: Collision Handling

```
function softCollision(softObject1, softObject2):
    Input  : softObject1, softObject2: The soft objects to
             check for collision
    Output: The updated soft objects after collision handling
    if collision detected between softObject1 and
      softObject2 using Gilbert-Johnson-Keerthi(GJK) then
        for vertex in collisionVertices do
            Apply force based on relative velocities and
              stiffness;
        end
        Calculate the new volume of softObject1 and
          softObject2;
        if volume changed then
            Apply volume conservation method to scale
              vertices;
        end
    end
    return softObject1, softObject2;
end
```

Algorithm 2: Collision Handling for Soft Objects

2.2 Mouse Interaction

When the left mouse button is clicked, the closest object to the mouse pointer is attached to the mouse and follow the mouse pointer's movement. The mouse button callback function is registered to detect the mouse click. Then, the closest point on an object to the mouse pointer is calculated when a click is detected. Finally in the simulation loop, the position of the chosen object is updated to follow the mouse.

```
function rigidCollision(rigidObject1, rigidObject2):
    Input  : rigidObject1, rigidObject2: The rigid objects to
             check for collision
    Output: The updated rigid objects after collision
             handling

    if distance between centers $<$ sum of radius then
        Compute unit collision direction vector;
        Swap velocities of rigidObject1 and rigidObject2
          along the collision direction;
    end
    return rigidObject1, rigidObject2;
end
```

Algorithm 3: Collision Handling for Rigid Objects

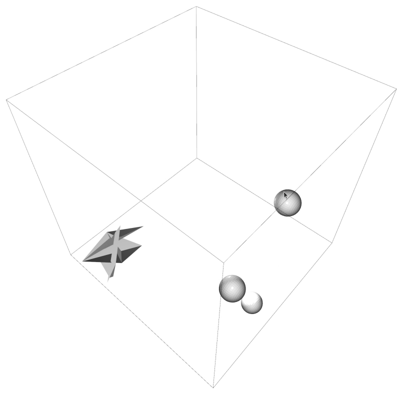

Figure 4: Mouse Interaction

3 Challenging creativity

3.1 Creative scene

The scene comprises a diverse array of objects, each possessing distinct physical characteristics, vigorously rebounding within the confines of a cubic space. Among them are rigid objects, taking the form of spheres, as well as a handful of custom-designed soft body shapes. Initially assuming the shape of a cube, the soft body object undergoes a transformation, deforming into a funny shape. Each object is granted a random position and velocity initially, while the force of gravity persists within the cube, prompting the objects to descend towards the lower reaches of the space.

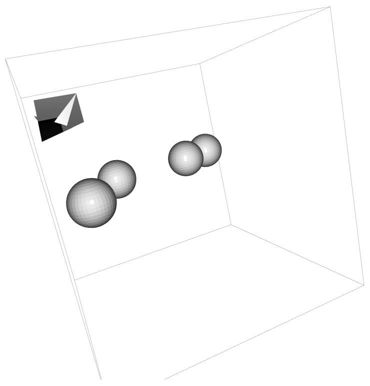

Figure 5: Creative Scene

4 Spatial Hashing for Large-scale Simulation

4.1 Implementation Details

Spatial hashing is an optimization technique used in large-scale simulations to reduce the computational cost of detecting collisions between objects.[4] The basic idea behind spatial hashing is to divide the simulation space into a grid of cells. Each cell is associated with a hash value. Objects in the simulation are associated with the cells that contain them, which can be determined based on their positions. The spatial hash is typically implemented as a hash map, where the keys are cell coordinates and the values are lists of objects.[5]

4.2 Time comparison

The running time of a simulation using spatial hashing scales significantly better with the population size compared to a simulation without spatial hashing. Without spatial hashing, the running time grows quadratically with the population size because each object potentially interacts with every other object.

Let n be the population size. Then, the running time of a simulation without spatial hashing is $O(n^2)$, because each object potentially interacts with every other object. On the other hand, the running time of a simulation with spatial hashing is typically sublinear, because each object only needs to check for interactions with objects in the same or adjacent cells.

For instance, in a test with population sizes ranging from 1 to 10000, the running time without spatial hashing increased much faster than the running time with spatial hashing as the population size increased. Thus, spatial hashing can significantly improve the performance of large-scale simulations.

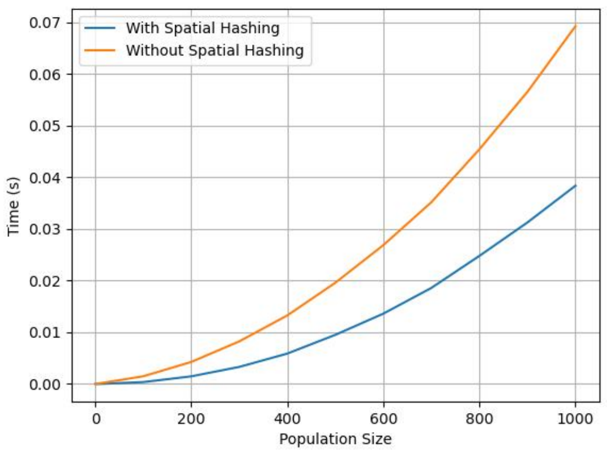

Figure 6: Time comparison of spatial hashing

Table 1: Running Time Comparison: Spatial Hashing vs. No Spatial Hashing

Pop.Size	**With(s)**	**Without (s)**
1	1.67e-06	3.79e-05
10	5.25e-06	9.94e-05
100	3.72e-04	1.55e-03
1000	3.88e-02	6.88e-02
2000	1.51e-01	2.66e-01
3000	3.36e-01	5.72e-01
4000	6.09e-01	1.02
5000	0.93	1.58
10000	3.87	6.11

References

[1] G. I. C. S. R. Fedkiw, “Volume conserving finite element simulations of deformable models,” 2007. Accessed: 2023-05-16.

[2] T. Chuang, “Design and qualitative/quantitative analysis of multi-agent spatial simulation library,” 2007. Accessed: 2023-05-14.

[3] P. Lindemann, “The gilbert-johnson-keerthi distance algorithm,” 2009. Accessed: 2023-05-16.

[4] M. T. B. H. M. M. D. P. M. Gross, “Optimized spatial hashing for collision detection of deformable objects,” 2003. Accessed: 2023-05-15.

[5] S. L. H. Hoppe, “Perfect spatial hashing,” 2006. Accessed: 2023-05-15.